The Formal Specification

and Derivation of CMOS-circuits

by

Rudolf H. Mak

85/01

January 1985

COMPUTING SCIENCE NOTES

Eindhoven University of Technology
Department of Mathematics and Computing Science
P.O. Box 513
5600 MB EINDHOVEN
The Netherlands
All rights reserved
editor: M.L. Potters

The formal specification and derivation of CMOS-circuits

Rudolf H. Mak

Department of Mathematics and Computing Science,

Eindhoven University of Technology,

P.O. Box 513, 5600 MB Eindhoven, The Netherlands.

Abstract

A programming notation for CMOS-circuits is given. With each circuit a Boolean expression is associated that specifies the logic properties of the circuit. Circuits are designed in a hierarchical fashion and rules are given to derive the logic properties of a composite circuit from the logic properties of its subcomponents. Combinational circuits and sequential circuits are treated in a uniform fashion.

1. Introduction

The task of designing a circuit is, or should become, very similar to the task of designing a program. Ideally, one would like to describe a circuit in some "high level" language, in which the designer needs to be concerned with the functional aspects of his design only, and is not burdened with the physics underlying the constituting components, nor with the problem of their layout on the chip. Hence, just like programs, we would like to be able to derive circuits from their formal specifications. We show that this can be achieved by postulating two rules, the substitution rule described in section 3, and the elimination rule described in section 4.

2. CMOS-circuits and their notation

A CMOS-circuit can conveniently be thought of as consisting of a collection of ports connected by a network of switches and wires. For a discussion of CMOS-switches we refer to [1]. Our interest is in circuits that consist of a hierarchy of components. In order to facilitate the reasoning about such circuits we shall require that the network connecting the ports of a component and its subcomponents consists solely of wires. Hence the switches become the basic components, that form the bottom level of the hierarchy. Components communicate signals through their ports. Notice that the same signal may be communicated through several distinct ports, which then have to be connected. This connection may exist either inside or outside the component. How many ports there are per signal, and whether connections are realized inside or outside the component are clearly concerns for an implementation. In this paper we shall not address this question, but we shall describe components in terms of signals.

We introduce a programming notation to specify and describe CMOS-components. In this notation a description of a component consists of

- a heading, stating the name of the component, and the names and types of the external signals by which the component communicates with its environment
- a local network, stating the subcomponents and a list of "connections" between the signals of the component, i.e. the external signals of the component and the external signals of its subcomponents, which are called the internal signals
- a Boolean expression, denoting the logic relation the component establishes between the external signals.

A specification of a component consists of a heading and a Boolean expression only. Notice that a specification describes the logic function of a component; the order in which the signals change their values is left unspecified.

The set of Boolean values is denoted by $B = \{zero, one\}$. The Boolean operators negation, conjunction, disjunction, equivalence, and implication are denoted by the symbols $'$, $\wedge$, $\vee$, $\equiv$, and $\Rightarrow$, respectively.

As an example we describe our basic components: the switches. There are two kinds of switches. A normally-off switch denoted by

```
com switch1 ( a , x : in , y : out) ;
    a ⟶ x = y
moc { a ∧ y  ≡  a ∧ x }
```

and a normally-on switch denoted by

```
com switch0 ( a , x : in , y : out) ;
    a' ⟶ x = y
moc { a' ∧ y  ≡  a' ∧ x }
```

There are two types of signals: input signals (indicated by in), and output signals (indicated by out). The local network of a switch is special, since it contains no subcomponents and states a conditional connection between the signals x and y . Switches are the only components with conditional connections in their local network. The shape of the local network for other components is described in the next section.

3. Substitution rule

For any component other than a switch the local network contains various subcomponents. The connection pattern between the ports is given by an equivalence relation upon the signals of which there are three kinds: (i) the external signals, (ii) the internal signals, (iii) the constant signals zero and one . The equivalence classes are called nets and all signals in a net are assumed to have the same value. This assumption is captured by the

<u>Substitution rule</u>

Let E be a Boolean expression and let p and q be two signals from the same net. Then E^p_q and E are equivalent, where E^p_q is the expression obtained from E when all occurrences of p are replaced by q .

The equivalence relation is denoted by the infix operator $=$, pronounced as "is connected with". Nets are denoted by listing a sufficient number of pairs of equivalent signals. For instance the net $\{a,b,c\}$ can be denoted by

(i) $a = b \text{ , } b = c$

(ii) $a = b \text{ , } a = c$

(iii) $a = c \text{ , } b = c$

(iv) $a = b \text{ , } b = c \text{ , } a = c$

Notice that a net specified by (i), for instance, may be realized by wires connecting the port(s) for signal a with the port(s) for signals b and c . Therefore we introduce yet another notation for nets, that avoids to suggest any implementation, and that is more concise. In this notation net $\{a,b,c\}$ is denoted by

(v) $a = b = c$

Thus the local network of a component consists of a list of declarations of subcomponents and a list of all nets.

The occurrence of two external input signals in the same net is forbidden, since it makes two signals of the environment equivalent; the occurrence of two external output signals in the same net indicates a superfluous output signal; the occurrence of two external signals of different type in the same net indicates a superfluous connection. Therefore we impose upon components the following

Syntactic restriction

Each net contains at most one external signal.

In the remainder of this section we demonstrate how the substitution rule is used to prove the correctness of components. Consider a selector specified by

com selector (a , x0 , x1 : in , y : out)

moc { $y \equiv a' \wedge x0 \vee a \wedge x1$ }

From the specification we derive by means of propositional calculus that a selector can be composed of two switches of opposite kind. In this and further derivations the equality sign between Boolean expressions means that the expressions immediate before and after the sign are equivalent. Between brackets a hint is given why this is so.

$y \equiv a' \wedge x0 \vee a \wedge x1$

= {propositional calculus}

$(a \vee (y \equiv a' \wedge x0 \vee a \wedge x1)) \wedge (a' \vee (y \equiv a' \wedge x0 \vee a \wedge x1))$

= {propositional calculus}

$(a' \wedge y \equiv a' \wedge (a' \wedge x0 \vee a \wedge x1)) \wedge$

$(a \wedge y \equiv a \wedge (a' \wedge x0 \vee a \wedge x1))$

= {propositional calculus}

$(a' \wedge y \equiv a' \wedge x0) \wedge (a \wedge y \equiv a \wedge x1)$

Internal signals of a component are denoted as follows. Let s be the name of a subcomponent, and e the name of one of its external signals. Then s.e is the name of the corresponding internal signal. With this notational convention we are able to give a program for the selector

```
com selector ( a , x0 , x1 : in , y : out) ;
    sub s0 : switch0 { s0.a' ∧ s0.y  ≡  s0.a' ∧ s0.x } ;
        s1 : switch1 { s1.a ∧ s1.y  ≡  s1.a ∧ s1.x } ;

    x0 = s0.x , x1 = s1.x ,
    a = s0.a = s1.a ,
    y = s0.y = s1.y

moc { y  ≡  a' ∧ x0  ∨  a ∧ x1 }
```

The correctness proof consists of an application of the substitution rule and the derivation given above, i.e.

$(s0.a' \wedge s0.y \equiv s0.a' \wedge s0.x) \wedge (s1.a \wedge s1.y \equiv s1.a \wedge s1.x)$

= {substitution rule}

$(a' \wedge y \equiv a' \wedge x0) \wedge (a \wedge y \equiv a \wedge x1)$

= {propositional calculus}

$y \equiv a' \wedge x0 \vee a \wedge x1$

Another example of a component that can be proved correct by means of the substitution rule is an inverter.

```
com inverter ( a : in , y : out) ;
    sub s : selector { s.y ≡ s.a' ∧ s.x0 ∨ s.a ∧ s.x1 } ;

    one = s.x0 , zero = s.x1 ,
    a = s.a , y = s.y

moc { y ≡ a' }
```

In the previous examples the substitution rule is sufficient to prove the correctness of components, since each net contains an external signal. In the case of components with nets that consist entirely of internal signals we need an additional rule.

4. Elimination rule

Consider the Boolean expression that specifies a component. It may be viewed as an equation in the external signals of the component. The solutions of this equation are called stable external signal configurations. The remaining configurations are called unstable external signal configurations. There is an obvious mechanistic appreciation of stable and unstable configurations. Any mechanism for a component that observes an unstable configuration shall try to reach a stable configuration by changing some of its output signals. Thereafter it remains in rest until it is brought into an unstable configuration by a change of an input signal initiated by its environment. From this mechanistic appreciation we conclude that for any assignment to the signals such that each subcomponent is in a stable configuration, the component itself is in a stable configuration. As a consequence of the substitution rule we only have to assign values to the external signals and one value per net that consists of internal signals only. Hence we introduce the following rule

Elimination rule

Let C be a component with $n \geq 1$ subcomponents specified by the Boolean expressions E_i , $0 \leq i < n$. Moreover, let there be $m \geq 0$ nets N_j , $0 \leq j < m$, with internal signals only. Then C satisfies the Boolean expression

$$(\exists\ p_0,\ldots,p_{m-1} : p_0,\ldots,p_{m-1} \in B : E)$$

where E is the conjunction of all E_i , with for $0 \leq j < m$ each signal in net N_j replaced by p_j .

We remark that in the case $m = 0$ the elimination rule yields the conjunction of all E_i , $0 \leq i < n$. As such it has already been used to prove the selector correct.

The following example illustrates the application of the elimination rule in a non-trivial case (i.e. $m > 0$).

```
com and ( a0 , a1 : in , y : out) ;
    sub s0 , s1 : selector { (s0.y  ≡  s0.a' ∧ s0.x0  ∨  s0.a ∧ s0.x1)
                           ∧ (s1.y  ≡  s1.a' ∧ s1.x0  ∨  s1.a ∧ s1.x1) } ;

    zero = s0.x0 = s1.x0 ,
    a0 = s0.a , a1 = s1.a ,
    one = s0.x1 , s0.y = s1.x1 , s1.y = y

moc { y  ≡  a0 ∧ a1 }
```

This component contains one net with internal signals only, viz. the net {s0.y,s1.x1} . Application of the elimination rule yields the Boolean expression

$$(\exists p : p \in B : (p \equiv s0.a' \wedge s0.x0 \vee s0.a \wedge s0.x1) \wedge \\ (s1.y \equiv s1.a' \wedge s1.x0 \vee s1.a \wedge p))$$

Moreover,

$(\exists p : p \in B : (p \equiv s0.a' \wedge s0.x0 \vee s0.a \wedge s0.x1) \wedge$
$\qquad (s1.y \equiv s1.a' \wedge s1.x0 \vee s1.a \wedge p))$

= {substitution rule}

$(\exists p : p \in B : (p \equiv a0) \wedge (y \equiv a1 \wedge p))$

= {predicate calculus}

$(\exists p : p \in B : p \equiv a0) \wedge (y \equiv a1 \wedge a0)$

= {predicate calculus}

$y \equiv a0 \wedge a1$

This completes the correctness proof for the and-component. In the next two sections we discuss some more examples.

5. A component with a precondition

Some components require a restriction on the input signals in order to meet their specification. An example of such a component is a Set-Reset flipflop. An SR-flipflop is specified by

```
com srff ( s , r : in , y , z : out)


moc { (z ≡ y') ∧ (s ⇛ y) ∧ (r ⇛ z) }
```

Observe that each stable external signal configuration satisfies $s \wedge r \equiv zero$. Since both s and r are input signals, we can prove the correctness of an SR-flipflop only under the restriction that the environment meets the specification $s \wedge r \equiv zero$. In our notation we add such a specification for the environment as a precondition to our component. We shall now prove the following version of an SR-flipflop to be correct.

```
{ s ∧ r ≡ zero }
com srff ( s , r : in , y , z : out) ;
    sub i0 , i1 : inverter { (i0.y ≡ i0.a') ∧ (i1.y ≡ i1.a') } ;
        s0 , s1 : selector { (s0.y ≡ s0.a' ∧ s0.x0 ∨ s0.a ∧ s0.x1)
                           ∧ (s1.y ≡ s1.a' ∧ s1.x0 ∨ s1.a ∧ s1.x1) } ;

    zero = s0.x1 = s1.x1 ,
    s = s0.a , s0.y = i0.a , i0.y = s1.x0 = y ,
    r = s1.a , s1.y = i1.a , i1.y = s0.x0 = z

moc { (z ≡ y') ∧ (s ⇛ y) ∧ (r ⇛ z) }
```

According to the elimination rule this component satisfies

$(\exists\ p,q : p,q \in B : (i0.y \equiv p') \wedge (i1.y \equiv q') \wedge$
$\qquad (p \equiv s0.a' \wedge s0.x0 \vee s0.a \wedge s0.x1) \wedge$
$\qquad (q \equiv s1.a' \wedge s1.x0 \vee s1.a \wedge s1.x1)\ )$

= {substitution rule}

$(\exists\ p,q : p,q \in B : (y \equiv p') \wedge (z \equiv q') \wedge$
$\qquad (p \equiv s' \wedge z) \wedge (q \equiv r' \wedge y)\ )$

= {predicate calculus}

$(\exists\ p,q : p,q \in B : (y \equiv p') \wedge (z \equiv q')\ ) \wedge$
$(y' \equiv s' \wedge z) \wedge (z' \equiv r' \wedge y)$

= {predicate calculus}

$(y' \equiv s' \wedge z) \wedge (z' \equiv r' \wedge y)$

= {predicate calculus}

$(y \vee (s' \wedge z)) \wedge (y' \vee s \vee z') \wedge (z \vee (r' \wedge y)) \wedge (z' \vee r \vee y')$

= {predicate calculus}

$(y' \vee z' \vee (s \wedge r)) \wedge (y \vee z) \wedge (y \vee s') \wedge (z \vee r')$

= {precondition $s \wedge r \equiv$ zero}

$((y' \vee z') \wedge (y \vee z)) \wedge (y \vee s') \wedge (z \vee r')$

= {predicate calculus}

$(z \equiv y') \wedge (s \Rrightarrow y) \wedge (r \Rrightarrow z)$

6. A recursively defined component

A tally circuit of order $n \geq 0$ is a component that has n inputs and $n+1$ outputs. The i-th output signal has value one if and only if precisely i input signals have value one. Formally a tally circuit of order n is a component that computes the function

$T_n : B^n \longrightarrow B^{n+1}$, defined by

$$T_0 = \text{one} \ ,$$

$$T_{n+1}(X_n) = \begin{cases} (T_n(X_{n-1}),\text{zero}) & \text{if } x_n = \text{zero} \\ (\text{zero},T_n(X_{n-1})) & \text{if } x_n = \text{one} \end{cases} \quad , \ n \geq 0$$

where X_{-1} is the 0-dimensional vector, and for $n \geq 0$, $X_n = (X_{n-1}, x_n)$ is an n+1-dimensional vector of Boolean values. Let y_i be the i-th coordinate of the vector $T_{n+1}(X_n)$, $0 \leq i \leq n+1$, and let t_j be the j-th coordinate of $T_n(X_{n-1})$, $0 \leq j \leq n$. Then

$$\begin{aligned} y_0 &= t_0 \wedge x_n' \vee \text{zero} \wedge x_n \ , \\ y_i &= t_i \wedge x_n' \vee t_{i-1} \wedge x_n \ , \quad 1 \leq i \leq n \ , \\ y_{n+1} &= \text{zero} \wedge x_n' \vee t_n \wedge x_n \ . \end{aligned}$$

Obviously a tally circuit of order $n+1$ can be composed of a tally circuit of order n and $n+2$ selectors, one for each output signal y_i , $0 \leq i \leq n+1$. With an extension of our notation, that allows parametrized components, "rows" of signals, "rows" of subcomponents, and a concise way to denote a large number of connections, we are able to denote this component by

```
com tally(0) ( y : [0..0]out) ;
    one = y[0]
moc { y ≡ T0 }
```

and for $n \geq 0$

```
com tally(n + 1) ( x : [0..n]in , y : [0..n + 1]out) ;
    sub t : tally(n) { t.y  ≡  $T_n$(t.x) } ;
        s : [0..n + 1]selector { (∀i : 0 ≤ i ≤ n + 1 :
            s[i].y  ≡  s[i].a' ∧ s[i].x0  ∨  s[i].a ∧ s[i].x1) } ;

    zero = s[0].x1 = s[n + 1].x0
    all i : 0..n - 1 : x[i] = t.x[i] lla ,
    all i : 0..n : t.y[i] = s[i].x0 = s[i + 1].x1 lla ,
    all i : 0..n + 1 : x[n] = s[i].a , y[i] = s[i].y lla

moc { y  ≡  $T_{n+1}$(x) }
```

The proof is left to the reader.

7. Concluding remarks

The notation in this paper is an extension of the notation for restoring logic circuitry in CMOS proposed in [1]. We therefore believe that, just as is demonstrated in [1] and also in [2], it is possible to design restoring logic circuits by imposing syntactic restrictions on the nets of components. Besides the switches and the selector all components in this paper are restoring according to the rules of [2].

The notation introduced in this paper is also a good starting point for the automatic generation of layouts (see [3]). Due to the hierarchical nature of the programs, simple placement and routing strategies should suffice to generate layouts with a high degree of regularity.

A circuit is best described by its behaviour, i.e. all possible sequences of signals it accepts and produces. This can be done for instance by means of traces [4] or by means of Petri nets [5].

The logic properties of a circuit can then be derived from its behaviour. Decomposition of circuits in terms of their behaviours, however, is much harder than decomposition on the logic level. We expect that logic decomposition can serve as a guide in decomposing the behaviours of circuits.

Acknowledgements

This work has benefitted from many discussions in the Eindhoven VLSI Club. The author wishes to thank its members for their comments and for providing a stimulating environment.

References

[1] Rem, M. and C. Mead, "A notation for designing restoring logic circuitry in CMOS", Proc. 2nd Caltech Conference on VLSI, ed. Seitz, C.L., California Institute of Technology, Pasadena, Calif., 1981.

[2] Rem, M., "On the design of restoring logic circuitry", Proc. Advanced Course on VLSI Architecture, eds. Randell and Treleaven, University of Bristol, Prentice Hall International, 1982.

[3] Lierop, M.L.P. van, "A flexible bottom-up approach for layout generation", THE-Memorandum, Eindhoven University of Technology, Eindhoven, 1984.

[4] Snepscheut, J.L.A. van de, "Trace theory and VLSI design", Ph.D. thesis,Eindhoven University of Technology, Eindhoven, 1983.

[5] Molnar, C.E. and T. Fang, "An asynchronous system design methodology", Technical Memorandum No. 287, Washington University, St Louis, Missouri, 1981.